

GNU tools

mostly for text processing

Morgan Goose <http://morgangoose.com>

January 2010

This work is licensed under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Follow along

I've got these slides up on my website, grab it with:

```
$ wget -r -p http://morgangoose.com/p/gnu_tools/{ \
  ,index.rst \
  ,files/}
```

or grab and untar the archive I made of this presentation with all of the files for the demo. http://morgangoose.com/p/gnu_tools.tar.gz

This is all under Creative Commons and all that well, what I myself wrote is

Keep these in mind

Stick to the basics

Man pages and help flags are your friends

Don't commit to changes until you know your cases are good

Remember Unix tool philosophy.

Which I always interpret as use many tools optimized for one thing chained together to make light work.

Commands I attempt to go over

generating text: find, lsof, netstat, ls, ps, pstree, w3m, du, whois, dig, pidof

processing text: sed, awk, xargs, grep, wc, tail, head, sort, uniq,

helping functions: pipes |, watch, cat, kill, pkill, killall, rm, mkdir, touch, for, !, !\$, !#, history

I will be going over bit and pieces of these, and some less than others. So if you find anything interesting, consult the man pages, or the internet, because there may be yet even cooler things about these commands than I mention.

Super quick intro to regex

"A regular expression (regex or regexp for short) is a special text string for describing a search pattern.

You can think of regular expressions as wildcards on steroids.

You are probably familiar with wildcard notations such as *.txt to find all text files in a file manager."

- regular-expressions.info

Some familiar examples

The wildcard character is one that you might already use on a daily basis..

For a list of every .tar.gz file in your home directory.

```
$ ls ~/.tar.gz
```

Or perhaps you need every dir with a specific file.

```
$ ls -d /home/*/.bash_rc
```

Useful regex command chars

\$ Match ends with

^ Match starts with

- Use a range of chars

* Repeats the previous item zero or more times

+ Repeats the previous item once or more

? Rule for match preceding this char is optional

Continued ...

. Matches any single character except line breaks

\s Shorthand for whitespace

\d Shorthand for digits

\ Suppress meaning

| An or statement for multiple matches

[] Surround character class

Some warm ups

Let's start using some regex in sed, and awk, with some premade files.

If you haven't already, you can grab them from:

http://morgangoose.com/p/gnu_tools/files/

Additional information is available for [awk](#) and [sed](#) on other sites.

A bit about sed

sed is a **S** tream **ED** itor.

In one pass it will take in a stream of text and apply the rules it has been given to the input, and either print the output or save it to a file.

It was developed from 1973 to 1974 as a Unix utility by Lee E. McMahon of Bell Labs,

Nice sed bits

Sed works almost exactly like vim for searching and replacing, just feed it some text and it goes to town:

```
$ sed '/find_me/p' sed.txt
```

```
$ sed 's/find_me/changed_to/g' sed.txt
```

```
$ sed -e's/rox[x]*/rox/g' -e's/changed_to/find_me/g' sed.txt
```

That all should look fairly familiar to a regular vim user. Regex usage and chaining of expressions may be new however.

More advanced stuff

The -i flag will make change inplace, and the -e flag will let you chain searches together

```
$ sed -i -e'/bass ackwards/d' sed.txt
```

```
$ sed -i '5d' sed.txt
```

```
$ sed '1,10 s/^deff/def/g' sed.txt
```

The deleting a specific line is a good tidbit to remember. It'll save a bit of time when you have unruly ssh keys, and errors, after an os reinstall.

Awk is short for awesome k.

"AWK is a language for processing files of text. A file is treated as a sequence of records, and by default each line is a record. [...]"

An AWK program is of a sequence of pattern-action statements.

AWK reads the input a line at a time. [scanning] for each pattern in the program, and [executes the associated action]"

- Alfred V. Aho

So while sed is great for stream manipulation, awk is more akin to a super powerful for loop.

Awk has super simple syntax, and built in field variables. It can use regex as well as make system calls.

It also let me avoid for loops. It is a personal thing, but also its harder to chain up more things afterwards with it's output. So I say stick to awk and skip having to remember the for loop syntax.

Some awk examples

Awk can get even deeper than what I am showing, look at the [awk](#) stuff there and also on the wikipedia article for some more help.

```
$ awk '{print $1}' awk.txt
```

```
$ awk 'BEGIN {x=35; printf("x = %x hex, %o octal.n",x,x)}'
```

```
$ awk -F: '{if ($3 > 300 && $3 < 550){print $1}}' passwd
```

```
$ awk '/class/{++cs} /def/{++df} END{print cs df};' awk3.txt
```

Awk as programming

Awk has 3 blocks, BEGIN, main loop(s), and then END block. All use {} to denote what they contain.

So a full setup looks like this:

```
BEGIN {}
{}
.
.
.
END{}
```

This next example should pique the interest or at least look familiar to any of you programmers that have dabbled in the functional realm.

Revisit that last example

The last example was a bit cramped. Spacing it out, we can begin to see the breakdown of an awk program a bit more. That and we can make it a bit nicer on the output.

script.awk

```
/class /{
  ++cs; split($2, n, "(");
  print "class found " n[1] "(" }

/def /{
  ++df; split($2, n, "(");
  print "func found " n[1] "(" }

END{
  print "\n-----"
  print "Classes: " cs;
  print "Functions: " df }
```

Awk scripts can be run by awk on the cli but using the -f flag to specify what script to use for processing its input. eg:

```
$ awk -f script.awk awk3.txt
```

Awk in a tight spot

Not only will awk handle most cases where you'd use a for loop, it will also work like cut, head, wc, or sed:

```
$ awk 'NR > 5 {print $2 ":" $1}' awk.txt
```

```
$ awk 'END{ print NR}' awk.txt
```

```
$ awk '!/Detroit/' awk2.txt
```

```
$ awk '$NF ~ /^set/' awk3.txt
```

This sort of overlapping functionality isn't exclusive to awk, but its a main reason as to why I choose to make it my heavy lifter.

I find it is easier to remember these, they are more legible, and I can fulfill my need to bum characters in my bash chains.

An extra bonus with awk is you can not only specify the field separator, but also the record separator. So if you had to have \0 null characters or / as your record separators you'd use:

```
$ awk 'BEGIN{ RS=","}{print}' awk2.txt
```

Find as a data source

First and foremost, find is awesome. Never doubt this. It can be slow, but it still is going to be a wonderful tool in your belt.

```
$ find ./ | grep "file_you.want"
```

```
$ find ./ -name "file_you.want"
```

```
$ find ./ -type f -name "^file" ! -name "*.tar.gz"
```

```
$ find /var/www/*/ -type f -name '/^[0-9].[0-9]+.php/'
```

Really and truly. If you deal with lots of files, usually that you didn't make. You need to read up on find. It will save you time.

I say this also because there are exec flags you can use with find so you can do crazy stuff like this

```
$ find . -name ".svn" -exec rm -rf {} \;
```

Now with the awesome

Not only can find do regex, and file type. It'll do just about everything you want in finding a list of files.

```
$ find . -type d -empty -delete
```

```
$ find /var/spool/mail/*/ -type d -size +40000k
```

```
$ find /var/repos/ -type d -path "**/.svn" -o -path "**/.git"
```

```
$ find /home/*/ -type d -perm 777
```

If you want to rock out the find -path flag more, you can start excluding as well, but this will create strange, to you, output. That is if you mix included paths and excluded paths.

This is because included only means just show these, exclude everything else. Where as excluded only mean show me everything but these. So when mixed, it is confused. You need to add in another exclude everything, usually, to get what you want.

Note that rsync also has path inclusion and exclusion, in this same manner.

We should grep around a bit

Grep is fun and useful, but less fun than everything else I've mentioned. So it gets neglected.

```
$ grep 222.126.11.3 /var/log/httpd/*
```

```
$ grep -n "def " grep.txt
```

```
$ grep -vE "*.zip" grep2.txt
```

```
$ grep -R -l "misspelling" ./files/
```

```
$ grep -A 5 "def " grep.txt
```

The l flag is telling grep to print matching files

The n flag is asking grep to print out the line numbers of the matches

The R flag for grep tells it to search recursively.

The E flag is just like the e flag for sed, and the v flag tells it to exclude matches.

Take a second for questions

So what makes sense? What doesn't?

Now for the rest...

Pipes, better than Mario's

Notice that until this point, we never used any pipes. That is something of a testament on how much one can do with the sed, awk, and find commands with just flags, and files.

This character, |, is going to be the glue for nearly everything cool, and helpful, that you'll want to do in the *nix universe.

A pipe takes the stdout of one program, and then feeds it to the stdin of another program. They link together commands end to end.

Commands in between the first and last, act as *filters* and transform data before being given to the final command.

As a general rule any of the commands that take files as input will work with a pipe.

Lets link some pipes

One can really link a lot together with pipes, and mostly bypass ever needing to use a bash for loop.

```
$ dmesg | less
```

```
$ lsof ~/ | awk '{print $1}' | sort | uniq -c | sort -n
```

```
$ pgrep bash | awk '{system("lsof " $1)}'
```

I'll show it below, but a series of pipes (tubes if you will), can get long. Longer than any terminal width you may have set. The solution to this is to make them multi-line using the \ character.

At the end a long line that you want to keep going, but just want to break off into the next row, tack that on the end and hit enter. Bash knows what to do, it just takes in more input, and waits to execute until the final carriage return wasn't preceded by the \ character.

Xargs doesn't argue

So you now know how to make huge lists. But we want arguments, that ideally would go great if sent to a helper function. Xargs will do this for you.

```
$ find ~/files/ -name "*.torrent" | xargs rm -f
```

To take on args containing spaces or newlines from find are handled with two args, one each on find and xargs.

```
$ find ~/windows\ 98/ -print0 | xargs -0 /bin/rm -f
```

There are more bits to xargs, but I rarely find a use for them. That however doesn't mean you won't, so take a gander at the man page.

Processes, processes, processes

Procs are a *nix mainstay, and just like files, you want to do crap with them.

Who is using perl, and what are they running?

```
$ ps -waxu | grep perl
```

If you like seeing how processes relate

```
$ pstree -paul | less
```

On to the killing

What good is it exactly to just know a process, lets mess with them a bit.

If you hate mono:

```
$ pkill -9 -u `whoami` mono.exe
```

sendmail must die, well restart

```
$ killall -HUP sendmail
```

Putting this to heavy use

Lets look for mbox files not compressed, over 50MB, sort them by size, and then make a neat looking output for them, telling me which email address is the most huge.

```
$ find /var/spool/ -type f -size +50000k \  
| grep -vE "\*.gz" \  
| xargs du \  
| sort -n \  
| awk '{split($NF, n, "/"); print n[4] ": " $1}'
```

The gz exclusion could have been done in find with the extra ! not, then the name flag with a regex like so:

```
$ find /var/spool/ -type f ! -name "*.gz"
```

I kept it out so the line would fit onto the slide.

Oh, who's using my webserver

```
$ netstat -netp \  
| awk '/:80/ {split($5, y, ":"); print y[1]}' \  
| sort \  
| grep -v `hostname -i` \  
| uniq -c \  
| sort -n \  
| awk '{if ($1 > 100) print}' \  
| awk '{system("iptables -A INPUT -s " $2 " -j DROP")}'
```

Not these guys.

Broken down, this takes the netstat output:

- looks for lines showing use on port 80
- grabs the ip from that line, and prints it
- sorts all of the outputted ip addresses
- greps out localhost's ip (may not work)
- makes the list unique and counts occurrences
- sort them again numerically (overkill, but good tracing)
- awk to weed out low usage
- awk to add an iptables rule to block them

What ports are showing

So what ports are open on your server

```
$ netstat -neptl
```

What sockets are open and what procs have them

```
$ lsof -i -n -P
```

Mailserver only

```
$ lsof -i :25 -n -P
```

One more awk tidbit

Holy hell, look what we can do with one awk trick.

```
$ lsof -i :25 -n \  
| awk '{split($(NF-1), n, ":"); print n[1]}' \  
| sort \  
| uniq -c \  
| sort -n
```

We now know what ip's on our server are sending mail, sorted by socket count.

Bash fun tricks

```
$ mkdir this_is_a_super_long_dir
```

Tab complete is cool and all, but try this afterwards.

```
$ cd !$
```

Or if you forgot the sudo:

```
$ yum install some_fun*
```

```
$ sudo !!
```

If you need to go back to that dir you just were in

```
$ cd -
```

Lazy completes

Since there is little difference in tarballs

```
$ mv /home/file.{tar.gz,tgz,tar.bz,tbz} /someplace/
```

```
$ touch foo.txt
```

Wanted bar.txt instead

```
$ ^foo^bar
```

Simple rename as done this way

```
$ mv move_me.{txt,rst}
```

You can get an idea how this works by

```
$ echo move_me.{txt,rst}
```

I forget about these a lot of the time. When applied, they do save time and keystrokes. So much so that I made you use them in the first slide, to grab this presentation.

This will also work for nesting dirs

```
$ mkdir -p somedir/{a,b,c}/more/{1,2,3}/final
```

Now its a nice tree of dirs, with a,b,c each with a more dir that has dirs 1,2,3 each with a final dir inside them.

History helps/hurts

History, usually, outputs all the commands you've run in a terminal.

You ran this huge awk/sed/find monstrosity sometime, don't remember it, but need to run it again:

```
$ history | grep -e"find"
```

```
$ !<number in history>
```

Note that if you must put a password into the cli, do a history, find the number then history -d #.

Extra tidbits

```
$ mkdir ~/start/
```

```
$ mkdir -p ~/start/and/now/continue/to/make/dirs/
```

```
$ watch 'uptime; pstree -p <pid>'
```

```
$ pstree -paul `pidof httpd`
```

I've used it a number of times in the presentation already, but the ` char surrounding some command(s), one can also use \$(command) to do the same thing in bash.

links used to make this presentation

- http://en.wikipedia.org/wiki/List_of_Unix_utilities
- <http://en.wikipedia.org/wiki/AWK>
- <http://www.regular-expressions.info/>
- <http://www.vectorsite.net/tsawk.html>
- <http://tldp.org/LDP/abs/html/>
- <http://www.grymoire.com/Unix/Sed.html>

For the slides' creation

- <http://docutils.sourceforge.net/docs/user/slide-shows.html>
- <http://hackmap.blogspot.com/2009/10/rst2s5-with-syntax-highlighting.html>